# V*ISTA* EXTENSIBLE MARKUP LANGUAGE (XML) PARSER

## Technical and User Documentation

## Patch XT*7.3*58

## Version 1.1

## February 2002

Department of Veterans Affairs
VHA OI System Design & Development (SD&D)
Information Infrastructure Service (IIS)

# Document Revision History

The following table displays the revision history for this document. Revisions to the documentation are based on Patch XT*7.3*58. This patch released the **V*IST*A** XML Parser to the field.

| Date | Revision | Description | Author |
|---|---|---|---|
| 02/07/02 | Patch XT*7.3*58 | Reviewed and edited documentation for **V*IST*A** standards and format. Added new chapter named "XML Parser Use Example." | Wally, Fort, Susan Strack, Office of Information, Oakland, CA |
| 05/18/00 | Version 1.1 | Initial documentation written for this product by Science Applications International Corporation (SAIC). | Science Applications International Corporation of San Diego, CA |

# Contents

# Figures

Table of Figures

# Introduction

The **V***IST***A** Extensible Markup Language (XML) Parser is a full-featured, validating XML parser written in the M programming language and designed to interface with the **V***IST***A** suite of M-based applications. It is not a standalone product. Rather, it acts as a server application that can provide XML parsing capabilities to any client application that subscribes to the application programmer interface (API) specification detailed in this document.

The **V***IST***A** XML Parser employs two very different API implementations. The first is an event-driven interface that is modeled after the widely used Simple API for XML (SAX) interface specification. In this implementation, a client application provides a special handler for each parsing event of interest. When the client invokes the parser, it conveys not only the document to be parsed, but also the entry points for each of its event handlers. As the parser progresses through the document, it invokes the client's handlers for each parsing event for which a handler has been registered.

The second API implementation is based on the World Wide Web Consortium (W3C's) Document Object Model (DOM) specification. This API, which is actually built on top of the event-driven interface, first constructs an in-memory model of the fully parsed document. It then provides methods to navigate through and extract information from the parsed document.

The choice of which API to employ is in part dependent on the needs of the application developer. The event-driven interface requires the client application to process the document in a strictly top-down manner. In contrast, the in-memory model provides the ability to move freely throughout the document and has the added advantage of ensuring that the document is well formed and valid before any information is returned to the client application.

The **V***IST***A** XML Parser employs an Entity Catalog to allow storage of external entities such as document type definitions. The Entity Catalog is a VA FileMan-compatible database and can be manipulated using the usual VA FileMan tools.

## Term Definitions and XML Parser Concept

To understand the terms used in this documentation and the concept of the operation of an XML Parser, please review the W3C Architecture Domain website, Extensible Markup Language (XML) page at: http://www.w3.org/XML/ .

# Orientation

This documentation uses several methods to highlight different aspects of the material. “Snapshots” of computer dialogue (or other online displays) are shown in a non-proportional font and enclosed within a box. User responses to on-line prompts are highlighted in boldface. Boldface is also used to highlight a descriptive word or sentence. The Return or Enter key is illustrated by the symbol **<Enter>** when displayed in computer dialogue and is included in examples only when it may be unclear to the reader that such a keystroke must be entered. The following example indicates that you should type two question marks followed by pressing the Return or Enter key when prompted to select an option:

```
Select Primary Menu option: ??
```

Figure 99: How to access online help

M code, variable names, acronyms, the formal name of options, actual field names, and file names are represented with all uppercase letters.

# Known Issues

The following are known issues in this version of the XML parser. Some of these are due to certain limitations of the M programming language.

Unlike languages like Java that have multiple character encoding support built-in, M does not recognize character encodings that do not incorporate the printable ASCII character subset. Thus, 16-bit character encodings such as Unicode are not supported. Fortunately, a large number of 8-bit character encodings do incorporate the printable ASCII character subset and can be parsed. Because of this limitation, the **V***IST***A** XML Parser will reject any documents with unsupported character encodings.

The current version of the **V***IST***A** XML Parser does not support retrieval of external entities using the HTTP or FTP protocols (or for that matter, any protocols other than the standard file access protocols of the underlying operating system). Client applications using the event-driven interface can intercept external entity retrieval by the parser and implement support for these protocols if desired.

The parser uses the Kernel function FTG^%ZISH for file access. This function reads the entire contents of a file into an M global. There are several nuances to this function that manifest themselves in parser operation:

1. Files are opened with a time-out parameter. If an attempt is made to access a non-existent file, there is a delay of a few seconds before the error is signaled.

2. Files are accessed in text mode. The result is that certain imbedded control characters are stripped from the input stream and never detected by the parser. Because these control characters are disallowed by XML, the parser will not report such documents as non-conforming.

3. A line feed / carriage return sequence at the end of a document is stripped and not presented to the parser. Only in rare circumstances would this be considered significant data, but in the strictest sense should be preserved.

The parser allows external entities to contain substitution text that in some cases would violate XML rules that state that a document must be conforming in the absence of resolving such references. In other words, XML states that a non-validating parser should be able to verify that a document is conforming without processing external entities. This restriction constrains how token streams can be continued across entities. The parser recognizes most, but not all, of these restrictions. The effect is that the parser is more lax in allowing certain kinds of entity substitutions.

Parsers vary in how they enforce whitespace that is designated as required by the XML specification. This parser will flag the absence of any required whitespace as a conformance error, even in situations where the absence of such whitespace would not introduce syntactic ambiguity. The result is that this parser will reject some documents that may be accepted by other parsers.

# Event-Driven Application Programmer Interface

The event-driven Application Programmer Interface (API) is based on the well-established Simple API for XML (SAX) interface employed by many XML parsers. This API, Figure 1, has a single method. (Figure 1 spans two pages.)

## EN^MXMLPRSE(DOC,CBK,OPT)

| Parameter | Type | Required | Description |
|---|---|---|---|
| **DOC** | String | Yes | This is either a closed reference to a global root containing the document or a filename and path reference identifying the document on the host system. If a global root is passed, the document must either be stored in standard FileMan word-processing format or may occur in sequentially numbered nodes below the root node. Thus, if the global reference is "^XYZ", the global must be of one of the following formats:<br>`^XYZ(1,0) = "LINE 1"`<br>`^XYZ(2,0) = "LINE 2" ...`<br>`or`<br>`^XYZ(1) = "LINE 1"`<br>`^XYZ(2) = "LINE 2"  ...` |
| **CBK** | Local array (by reference) | No | This is a local array, passed by reference that contains a list of parse events and the entry points for the handlers of those events. The format for each entry is:<br>`CBK(<event type>) = <entry point>`<br>The entry point must reference a valid entry point in an existing M routine and should be of the format *tag^routine*. The entry should not contain any formal parameter references. The application developer is responsible for ensuring that the actual entry point contains the appropriate number of formal parameters for the event type. For example, client application might register its STARTELEMENT event handler as follows:<br>`CBK("STARTELEMENT") = "STELE^CLNT"`<br>The actual entry point in the CLNT routine must include two formal parameters as in the example:<br>`STELE(ELE,ATR)  <handler code>`<br>For the types of supported events and their required parameters, see the discussion on the pages that follows. |

| Parameter | Type | Required | Description |
|---|---|---|---|
| **OPT** | String | No | This is a list of option flags that control parser behavior. Recognized option flags are:<br>• *W* = Do not report warnings to the client.<br>• *V* = Do not validate the document. If specified, the parser only checks for conformance.<br>• *0* = Terminate parsing on encountering a warning.<br>• *1* = Terminate parsing on encountering a validation error. (By default, the parser terminates only when a conformance error is encountered.) |

Figure 1: EN^MXMLPRSE(DOC,CBK,OPT) – Event-Driven API based on SAX interface

The **V*IST*A** XML Parser recognizes the event types Figure 2. (Figure 2 spans two pages.)

| Event Type | Parameter(s) | Description |
|---|---|---|
| **STARTDOCUMENT** | None | Notifies the client that document parsing has commenced. |
| **ENDDOCUMENT** | None | Notifies the client that document parsing has completed. |
| **DOCTYPE** | ROOT<br>PUBID<br>SYSID | Notifies the client that a DOCTYPE declaration has been encountered. The name of the document root is given by *ROOT*. The public and system identifiers of the external document type definition are given by *PUBID* and *SYSID*, respectively. |
| **STARTELEMENT** | NAME<br>ATTRLIST | An element (tag) has been encountered. The name of the element is given in *NAME*. The list of attributes and their values is provided in the local array *ATTRLST* in the format:<br>`ATTRLST(<name>) = <value>` |
| **ENDELEMENT** | NAME | A closing element (tag) has been encountered. The name of the element is given in *NAME*. |
| **CHARACTERS** | TEXT | Non-markup content has been encountered. *TEXT* contains the text. Line breaks within the original document are represented as carriage return/line feed character sequences. The parser does not necessarily pass an entire line of the original document to the client with each event of this type. |
| **PI** | TARGET<br>TEXT | The parser has encountered a processing instruction. *TARGET* is the target application for the processing instruction. *TEXT* is a local array containing the parameters for the instruction. |
| **EXTERNAL** | SYSID<br>PUBID<br>GLOBAL | The parser has encountered an external entity reference whose system and public identifiers are given by *SYSID* and *PUBID*, respectively. If the event handler elects to retrieve the entity rather than allowing the parser to do so, it should pass the global root of the retrieved entity in the *GLOBAL* parameter. If the event handler wishes to suppress retrieval of the entity altogether, it should set both *SYSID* and *PUBID* to null. |

| Event Type | Parameter(s) | Description |
|---|---|---|
| **NOTATION** | NAME<br>SYSID<br>PUBIC | The parser has encountered a notation declaration. The notation name is given by *NAME*. The system and public identifiers associated with the notation are given by *SYSID* and *PUBIC*, respectively. |
| **COMMENT** | TEXT | The parser has encountered a comment. *TEXT* is the text of the comment. |
| **ERROR** | ERR | The parser has encountered an error during the processing of a document. *ERR* is a local array containing information about the error. The format is:<br>• ERR("SEV") = Severity of the error where 0 is a warning, 1 is a validation error, and 2 is a conformance error.<br>• ERR("MSG") = Brief text description of the error.<br>• ERR("ARG") = The token value the triggered the error (optional).<br>• ERR("LIN") = The number of the line being processed when the error occurred.<br>• ERR("POS") = The character position within the line where the error occurred.<br>• ERR("XML") = The original document text of the line where the error occurred. |

Figure 2: Event types recognized by the **V***IST***A** XML Parser

A sample client of the event-driven API is provided in the routine MXMLTEST. This routine has an entry point EN(DOC,OPT), where DOC and OPT are the same parameters as described above in Figure 1 for the parser entry point. This sample application simply prints a summary of the parsing events as they occur.

# In-Memory Document Application Programmer Interface

This Application Programmer Interface (API) is based on the W3C's Document Object Model (DOM) specification. It first builds an "in-memory" image of the fully parsed and validated document and then provides a set of methods to permit structured traversal of the document and extraction of its contents. This API is actually layered on top of the event-driven API. In other words, it is actually a client of the event-driven API that in turn acts as a server to another client application.

The document image is represented internally as a tree with each node in the tree representing an element instance. Attributes (names and values), non-markup text, and comment text may be associated with any given node. For example, in Figure 3 the XML document on the left is represented by the tree structure on the right.

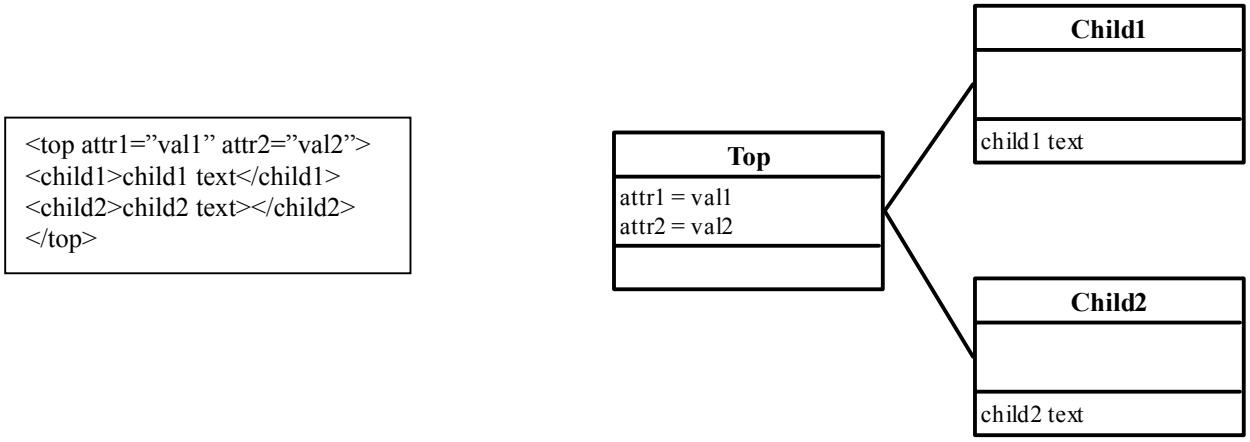


Figure 3: XML document (left) – Tree structure diagram (right)

The supported methods are documented on the pages that follow.

# $$EN^MXMLDOM

This is the entry point to perform initial processing of the XML document. The client application must first call this entry point to build the in-memory image of the document before the remaining methods can be applied. The return value is a handle to the document instance that was created and is used by the remaining API calls to identify a specific document instance. The parameters for this entry point are listed in Figure 4 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **DOC** | String | Yes | Either a closed reference to a global root containing the document or a filename and path reference identifying the document on the host system. If a global root is passed, the document must either be stored in standard FileMan word-processing format or may occur in sequentially numbered nodes below the root node. Thus, if the global reference is "^XYZ", the global must be of one of the following formats:<br><br>`^XYZ(1,0) = "LINE 1"`<br>`^XYZ(2,0) = "LINE 2" ...`<br><br>`or`<br><br>`^XYZ(1) = "LINE 1"`<br>`^XYZ(2) = "LINE 2" ...` |
| **OPT** | String | No | • *W* = Do not report warnings to the client.<br>• *V* = Do not validate the document. If specified, the parser only checks for conformance.<br>• *0* = Terminate parsing on encountering a warning.<br>• *1* = Terminate parsing on encountering a validation error. (By default, the parser terminates only when a conformance error is encountered.) |
| **Return value** | Integer | - | Returns a nonzero handle to the document instance if parsing completed successfully, or zero otherwise. This handle is passed to all other API methods to indicate which document instance is being referenced. This allows for multiple document instances to be processed concurrently. |

Figure 4: $$EN^MXMLDOM – Perform initial processing of XML document

# DELETE^MXMLDOM

This entry point deletes the specified document instance. A client application should always call this entry point when finished with a document instance. The parameter for this API is listed in Figure 5 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |

Figure 5: DELETE^MXMLDOM – Delete specified document instance

# $$NAME^MXMLDOM

This entry point returns the name of the element at the specified node within the document parse tree. The parameters for this API are listed in Figure 6 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node whose associated element name is being retrieved. |
| **Return Value** | String | - | The name of the element associated with the specified node. |

Figure 6: $$NAME^MXMLDOM – Return element name at specified node in document parse tree

# $$CHILD^MXMLDOM

Returns the node of the first or next child of a given parent node, or 0 if there are none remaining. The parameters for this API are listed in Figure 7 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **PARENT** | Integer | Yes | The node whose children are being retrieved. |
| **CHILD** | Integer | No | If specified, this is the last child node retrieved. The function will return the next child in the list. If the parameter is zero or missing, the first child is returned. |
| **Return Value** | Integer | - | The next child node or zero if there are none remaining. |

Figure 7: $$CHILD^MXMLDOM – Return parent node's first or next child. 0 if none remaining

# $$SIBLING^MXMLDOM

Returns the node of the specified node's immediate sibling, or 0 if there is none. The parameters for this API are listed in Figure 8 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node in the document tree whose sibling is being retrieved. |
| **Return Value** | Integer | - | The node corresponding to the immediate sibling of the specified node, or zero if there is none. |

Figure 8: $$SIBLING^MXMLDOM – Return specified node's immediate sibling. 0 if none remaining

# $$PARENT^MXMLDOM

Returns the parent node of the specified node, or 0 if there is none. The parameters for this API are listed in Figure 9 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node in the document tree whose parent is being retrieved. |
| **Return Value** | String | - | The parent node of the specified node, or zero if there is no parent. |

Figure 9: $$PARENT^MXMLDOM – Return specified node's parent node. 0 if none remaining

# TEXT^MXMLDOM or $$TEXT^MXMLDOM

Extracts non-markup text associated with the specified node. The parameters for this API are listed in Figure 10 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node in the document tree that is being referenced by this call. |
| **TEXT** | String | Yes | This parameter must contain a closed local or global array reference that is to receive the text. The specified array is deleted before being populated. |
| **Return Value** | Boolean | - | If called as an extrinsic function, the return value is true if text was retrieved, or false if not. |

Figure 10: TEXT^MXMLDOM or $$TEXT^MXMLDOM – Extract specified node's non-markup text

# CMNT^MXMLDOM or $$CMNT^MXMLDOM

Extracts comment text associated with the specified node. The parameters for this API are listed in Figure 11 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node in the document tree that is being referenced by this call. |
| **TEXT** | String | Yes | This parameter must contain a closed local or global array reference that is to receive the text. The specified array is deleted before being populated. |
| **Return Value** | Boolean | - | If called as an extrinsic function, the return value is true if text was retrieved, or false if not. |

Figure 11: CMNT^MXMLDOM or $$CMNT^MXMLDOM – Extract specified node's comment text

# $$ATTRIB^MXMLDOM

Retrieves the first or next attribute associated with the specified node. The parameters for this API are listed in Figure 12 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node whose attribute name is being retrieved. |
| **ATTRIB** | String | No | The name of the last attribute retrieved by this call. If null or missing, the first attribute associated with the specified node is returned. Otherwise, the next attribute in the list is returned. |
| **Return Value** | String | - | The name of the first or next attribute associated with the specified node, or null if there are none remaining. |

Figure 12: $$ATTRIB^MXMLDOM – Retrieve specified node's first or next attribute

# $$VALUE^MXMLDOM

Retrieves the value associated with the named attribute. The parameters for this API are listed in Figure 13 by type, requirement (yes or no), and description.

| Parameter | Type | Required | Description |
|---|---|---|---|
| **HANDLE** | Integer | Yes | The value returned by the $$EN^MXMLDOM call that created the in-memory document image. |
| **NODE** | Integer | Yes | The node whose attribute value is being retrieved. |
| **ATTRIB** | String | No | The name of the attribute whose value is being retrieved by this call. |
| **Return Value** | String | - | The value associated with the specified attribute. |

Figure 13: $$VALUE^MXMLDOM – Retrieve value associated with named attribute

# Entity Catalog

The entity catalog is used to store external entities and their associated public identifiers. When the XML parser encounters an external entity reference with a public identifier, it first looks for that public identifier in the entity catalog. If it finds the entity, it retrieves its value. Otherwise, it attempts to retrieve the entity value using the system identifier. The problem with using system identifiers is that they often identify resources that may have been relocated since the document was authored. (This is analogous to the problem with broken links in HTML documents.) Using public identifiers and an entity catalog allows one to build a collection of commonly used and readily accessible external entities (e.g., external document type definitions).

## XML ENTITY CATALOG (#950)

The entity catalog is a VA FileMan-compatible file that is very simple in structure:

| Field # | Field Name | Datatype | Description |
|---|---|---|---|
| **.01** | ID | Free text (1-250) | The public identifier associated with this entity. |
| **1** | VALUE | Word Processing | The text associated with the entity. |

Figure 14: XML ENTITY CATALOG file (#950) – Stores external entities and assoc public identifiers

# V*ISTA* XML Parser Usage Example

This is a simple example of how to use the **V***IST***A** XML Parser with an XML document (file). The XML file contains a parent node named BOOKS. Nested within that parent node are child nodes named TITLE and AUTHOR.

Remember that the:

- Parent node is the node whose child nodes are being retrieved.
- Child node, if specified, is the last child node retrieved. The function will return the next child in the list. If the parameter is zero or missing, the first child is returned.

## Create an XML File

```
^TMP($J,1) = <?xml version='1.0'?>
^TMP($J,2) = <!DOCTYPE BOOK>
^TMP($J,3) = <BOOK>
^TMP($J,4) = <TITLE>Design Patterns</TITLE>
^TMP($J,5) = <AUTHOR>Gamma</AUTHOR>
^TMP($J,6) = <AUTHOR>Helm</AUTHOR>
^TMP($J,7) = <AUTHOR>Johnson</AUTHOR>
^TMP($J,8) = <AUTHOR>Vlissides</AUTHOR>
^TMP($J,9) = </BOOK>
```

Figure 15: **V***IST***A** XML Parser Use Example – Create XML File

## Invoke Simple API for XML (SAX) Interface

```
D EN^MXMLTEST($NA(^TMP($J)),"V")  <Enter>
```

Figure 16: **V***IST***A** XML Parser Use Example – Invoke SAX Interface

. . . Now see what happens.

# Check Document Object Model (DOM) Interface

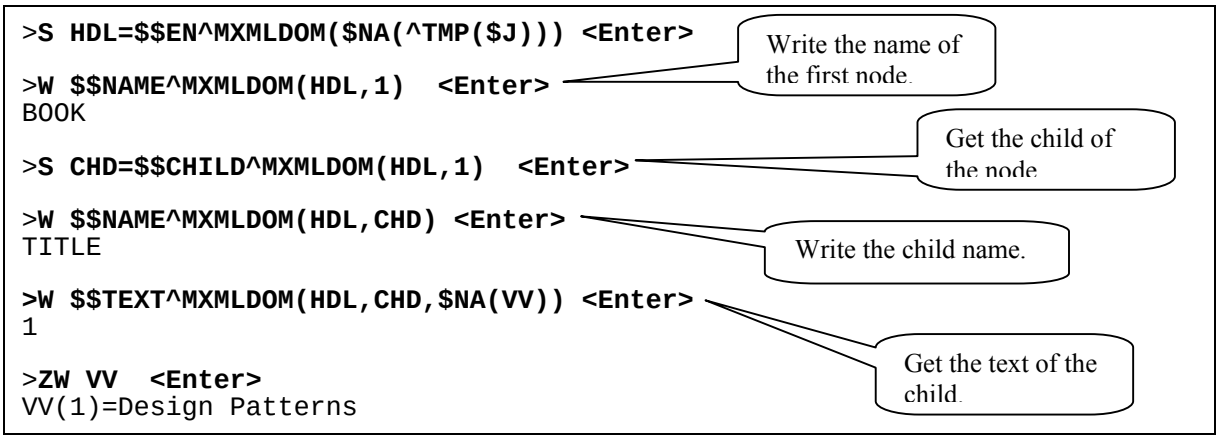


Figure 17: **V*IST*A** XML Parser Use Example – Check DOM Interface

# List All Sibling Nodes

```
>S CHD=$$CHILD^MXMLDOM(HDL,1)  <Enter>

>S SIB=CHD  <Enter>

>F  S SIB=$$SIBLING^MXMLDOM(HDL,SIB) Q:SIB'>0  W
!,SIB,?4,$$NAME^MXMLDOM(HDL,SIB)  <Enter>
3   AUTHOR
4   AUTHOR
5   AUTHOR
6   AUTHOR
>
```

Figure 18: **V*IST*A** XML Parser Use Example – List Sibling Nodes

# Index

## W



## X